AVF Control Number: AVF-VSR-193.0988
88-04-15-INT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880617W1.09115
Intermetrics, Inc.
UTS Ada Real-Time Compiler, Version 202.35
IBM 3083 (S/370)

Completion of On-Site Testing:
27 June 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETEING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Ada Compiler Validation Summary Report: Intermetrics, Inc., UTS Ada Real-Time Compiler, Version 202.35, IBM 3083 (S/370) (Host and Target). (880617W1.09115)

5. TYPE OF REPORT & PERIOD COVERED
27 June 1988 to 27 June 1989

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE
27 June 1988

13. NUMBER OF PAGES
50 p.

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)
Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)
UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)
Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
UTS Ada Real-Time Compiler, Version 202.35, Intermetrics, Inc., Wright-Patterson Air Force Base, IBM 3083 (S/370) under UTS, Version 2.3 (Host and Target), ACVC 1.9.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

Ada Compiler Validation Summary Report:

Compiler Name: UTS Ada Real-Time Compiler, Version 202.35

Certificate Number: 880617W1.09115

Host:	Target:
IBM 3083 (S/370) under UTS, Version 2.3	IBM 3083 (S/370) under UTS, Version 2.3

Testing Completed 27 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

John F. Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1-2
1.2 USE OF THIS VALIDATION SUMMARY REPORT 1-2
1.3 REFERENCES 1-3
1.4 DEFINITION OF TERMS 1-4
1.5 ACVC TEST CLASSES 1-5

CHAPTER 2 CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED 2-1
2.2 IMPLEMENTATION CHARACTERISTICS 2-2

CHAPTER 3 TEST INFORMATION

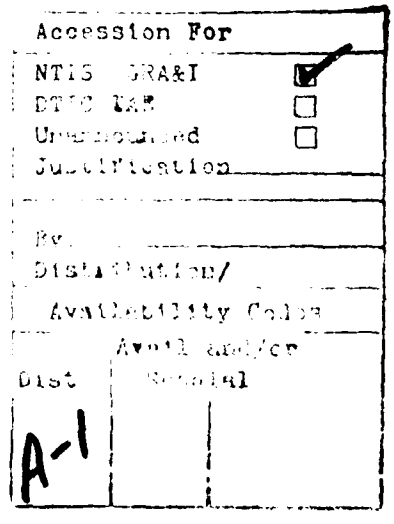
3.1 TEST RESULTS 3-1
3.2 SUMMARY OF TEST RESULTS BY CLASS 3-1
3.3 SUMMARY OF TEST RESULTS BY CHAPTER 3-2
3.4 WITHDRAWN TESTS 3-2
3.5 INAPPLICABLE TESTS 3-2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS . . 3-4
3.7 ADDITIONAL TESTING INFORMATION 3-5
3.7.1 Prevalidation 3-5
3.7.2 Test Method 3-5
3.7.3 Test Site 3-6

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B APPENDIX F OF THE Ada STANDARD

APPENDIX C TEST PARAMETERS

APPENDIX D WITHDRAWN TESTS



CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 27 June 1988 at Intermetrics, Inc., Cambridge MA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect

because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: UTS Ada Real-Time Compiler, Version 202.35

ACVC Version: 1.9

Certificate Number: 880617W1.09115

Host Computer:

Machine:	IBM 3083 (S/370)
Operating System:	UTS, Version 2.3
Memory Size:	24 megabytes

Target Computer:

Machine:	IBM 3083 (S/370)
Operating System:	UTS, Version 2.3
Memory Size:	24 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 17 levels and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. Block nesting is not supported to 65 levels. (See tests D55A03A..H (8 tests), D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64-bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined type SHORT_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

  The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

  The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

  The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

No exception is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

No exception is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is

compatible with the target's subtype. (See test C52013A.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

  In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

  All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

  For this implementation:

  An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

  Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

  Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

  Enumeration representation clauses for derived types are not supported. (See tests C35508I..J and C35508M..N.)

  Length clauses with SIZE specifications for enumeration types are not supported. (See test A39005B.)

  Length clauses with STORAGE_SIZE specifications for access types are not supported. (See tests A39005C and C87B62B.)

  Length clauses with STORAGE_SIZE specifications for task types are not supported. (See tests A39005D and C87B62D.)

  Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

  Record representation clauses with an alignment clause are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is supported for procedures and functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file can be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files and temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Body and subunits of a generic unit must be in the same compilation as the specification if instantiations precede them. (See tests CA2009C and CA2009F.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 267 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 5 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	103	1046	1607	12	14	46	2828
Inapplicable	7	5	246	5	4	0	267
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	491	528	237	166	98	137	327	135	36	234	3	246	2828
Inapplicable	14	81	146	11	0	0	6	0	2	0	0	0	7	267
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	A35902C	C35502P
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	BC3105A	CC1311B	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 267 tests were inapplicable for the reasons indicated:

- C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for derived types. These clauses are not supported by this compiler.
- C35702B uses LONG_FLOAT which is not supported by this implementation.

- A39005B and C87B62A use length clauses with SIZE specifications for derived integer types or for enumeration types which are not supported by this compiler.

- A39005C..D (2 tests), C87B62B and C87B62D use length clauses with STORAGE_SIZE specifications for access types or for task types which are not supported by this implementation.

- A39005E and C87B62C use length clauses with SMALL specifications which are not supported by this implementation.

- A39005G uses a record representation clause with an alignment clause which is not supported by this compiler.

- The following tests use SHORT_INTEGER, which is not supported by this compiler:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

- The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- D55A03E..H (4 tests) use more than 17 levels of loop nesting which exceeds the capacity of the compiler.

- D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- CA2009C and CA2009F compile generic subunits in separate compilation files. For this implementation, the body and subunits of a generic unit must be in the same compilation as the specification if instantiations precede them.

- AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with *unconstrained* array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

- AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

- CE3111B assumes that if the same external file is open for both reading and writing, then characters written may be immediately re-read, without a new-line/reset/close separating the read and write. This implementation buffers output and requires that a reset be issued between writing and reading from the same external file, if the read wants to be sure to see the effect of the write.

- The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 5 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B49003A B49005A BA1101C4 BC3205D BC3604A

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the UTS Ada Real-Time Compiler, Version 202.35, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the UTS Ada Real-Time Compiler, Version 202.35, using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of an IBM 3083 (S/370) operating under UTS, Version 2.3.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled on the IBM 3083 (S/370), and all executable tests were linked and run on the IBM 3083 (S/370). Results were printed from the host computer.

The compiler was tested using command scripts provided by Intermetrics, Inc. and reviewed by the validation team. The compiler was tested using all default switch settings.

The Intermetrics UTS Ada Compiler was invoked using the "aiecomp" command. A program build (link) was done using "aiebuild." A program library was created using the "cre_cat" command of PLM, the program library manager. The Report Package and CHECK_FILE were compiled into a base program library before the start of prevalidation testing. Then, each new program library that is created has access to these packages. This avoids their recompilation each time a new library is created.

The validation testing was controlled using a batch facility on the IBM 3083. Automated testing tools were used to ensure that the same environment was used for each ACVC test by reinitializing the library before each ACVC test.

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Intermetrics, Inc., Cambridge MA and was completed on 27 June 1988. The IBM 3083 (S/370) was not dedicated to the testing effort.

APPENDIX A

DECLARATION OF CONFORMANCE

Intermetrics, Inc. has submitted the following Declaration of Conformance concerning the UTS Ada Real-Time Compiler, Version 202.35.

DECLARATION OF CONFORMANCE

Compiler Implementor: Intermetrics, Inc.
Ada Validation Facility: Ada Validation Facility, ASD/SCEL,
Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: UTS Ada Real-Time Compiler Version: 202.35
Host Architecture ISA: IBM 3083 (S/370) OS&VER #: UTS, 2.3
Target Architecture ISA: IBM 3083 (S/370) OS&VER #: UTS, 2.3

Implementor's Declaration

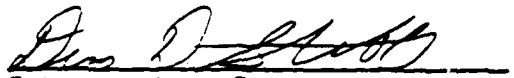
I, the undersigned, representing Intermetrics, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Intermetrics, Inc. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

Date: 6/20/88

Intermetrics, Inc.
Dennis Struble, Deputy General Manager

Owner's Declaration

I, the undersigned, representing Intermetrics, Inc., take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Date: 6/20/88

Intermetrics, Inc.
Dennis Struble, Deputy General Manager

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the UTS Ada Real-Time Compiler, Version 202.35, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -2_147_483_647 .. 2_147_483_647;

     type FLOAT is digits 15 range 2.0**(-205) .. (1.0-2**(-51))*2.0**204;
     type SHORT_FLOAT is digits 6 range 2.0**(-85) .. 2.0**84;

     type DURATION is delta 2.0**(-14) range -86_400 .. 86_400;

     ...

end STANDARD;
```

Appendix F. IMPLEMENTATION DEPENDENCIES

This section constitutes Appendix F of the Ada LRM for this implementation. Appendix F from the LRM states:

> *The Ada language allows for certain machine-dependencies in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependencies correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in Chapter 13, and certain allowed restrictions on representation clauses.*
>
> *The reference manual of each Ada implementation must include an appendix (called Appendix F) that describes all implementation-dependent characteristics. The Appendix F for a given implementation must list in particular:*
>
> 1. *The form, allowed places, and effect of every implementation-dependent pragma.*
> 2. *The name and the type of every implementation-dependent attribute.*
> 3. *The specification of the package SYSTEM (see 13.7).*
> 4. *The list of all restrictions on representation clauses (see 13.1).*
> 5. *The conventions used for any implementation-generated name denoting implementation-dependent components (see 13.4).*
> 6. *The interpretation of expressions that appear in address clauses, including those for interrupts (see 13.5).*
> 7. *Any restriction on unchecked conversions (see 13.10.2).*
> 8. *Any implementation-dependent characteristics of the input-output packages (see 14).*

In addition, the present section will describe the following topics:

9. Any implementation-dependent rules for termination of tasks dependent on library packages (see 9.4:13).
10. Other implementation dependencies.
11. Compiler capacity limitations.

F.1 Pragmas

This section describes the form, allowed places, and effect of every implementation-dependent pragma.

F.1.1 Pragmas LIST, PAGE, PRIORITY

Pragmas LIST, PAGE, and PRIORITY are supported exactly in the form, in the allowed places, and with the effect as described in the LRM.

F.1.2 Pragma SUPPRESS

Form: Pragma SUPPRESS (identifier)
where the identifier is that of the check that can be omitted. This is as specified in LRM B(14), except that suppression of checks for a particular name is not supported. The name clause (ON=>name), if given, causes the entire pragma to be ignored.

The suppression of the following run-time checks, which correspond to situations in which the exceptions CONSTRAINT_ERROR, STORAGE_ERROR, or PROGRAM_ERROR may be raised, are supported:

ACCESS_CHECK
DISCRIMINANT_CHECK
INDEX_CHECK
LENGTH_CHECK
RANGE_CHECK
STORAGE_CHECK
ELABORATION_CHECK

The checks which correspond to situations in which the exception NUMERIC_ERROR may be raised occur in the hardware and therefore pragma SUPPRESS of DIVISION_CHECK and OVERFLOW_CHECK are not supported.

Allowed Places: As specified in LRM B(14) : SUPPRESS.

Effect: Permits the compiler not to emit code in the unit being compiled to perform various checking operations during program execution. The supported checks have the effect of suppressing the specified check as described in the LRM. A pragma SUPPRESS specifying an unsupported check is ignored.

F.1.3 Pragma SUPPRESS_ALL

Form: Pragma SUPPRESS_ALL

Allowed Places: As specified in LRM B(14) for pragma SUPPRESS.

Effect: The implementation-defined pragma SUPPRESS_ALL has the same effect as the specification of a pragma SUPPRESS for each of the supported checks.

F.1.4 Pragma INLINE

Form: Pragma INLINE (SubprogramNameCommaList)

Allowed Places: As specified in LRM B(4) : INLINE.

Effect: If the subprogram body is available, and the subprogram is not recursive, the code is expanded in-line at every call site and is subject to all optimizations.

The stack-frame needed for the elaboration of the inline subprogram will be allocated as a temporary in the frame of the containing code.

Parameters will be passed properly, by value or by reference, as for non-inline subprograms. Register-saving and the like will be suppressed. Parameters may be stored in the local stack-frame or held in registers, as global code generation allows.

Exception-handlers for the INLINE subprogram will be handled as for block-statements.

Use: This pragma is used either when it is believed that the time required for a call to the specified routine will in general be excessive (this for frequently called subprograms) or when the average expected size of expanded code is thought to be comparable to that of a call.

F.1.5 Pragma INTERFACE

Form: Pragma INTERFACE (language_name, subprogram_name)
where the language_name must be an enumeration value of the type SYSTEM.Supported_Language_Name (see Package SYSTEM below).

Allowed Place: As specified in LRM B(5) : INTERFACE.

Effect: Specifies that a subprogram will be provided outside the Ada program library and will be callable with a specified calling interface. Neither an Ada body nor an Ada body_stub may be provided for a subprogram for which INTERFACE has been specified.

Use: Use with a subprogram being provided via another programming language and for which no body will be given in any Ada program. See

also the LINK_NAME pragma.

F.1.6 Pragma LINK_NAME

Form: Pragma LINK_NAME (subprogram_name, link_name)

Allowed Places: As specified in LRM B(5) for pragma INTERFACE.

Effect: Associates with subprogram subprogram_name the name link_name as its entry point name.

Syntax: The value of link_name must be a character string literal.

Use: To allow Ada programs, with help from INTERFACE pragma, to reference non-Ada subprograms. Also allows non-Ada programs to call specified Ada subprograms.

F.1.7 Pragma CONTROLLED

Form: Pragma CONTROLLED (AccessTypeName)

Allowed Places: As specified in LRM B(2) : CONTROLLED.

Effect: Ensures that heap objects are not automatically reclaimed. Since no automatic garbage collection is is ever performed, this pragma currently has no effect.

F.1.8 Pragma PACK

Form: Pragma PACK (type_simple_name)

Allowed Places: As specified in LRM 13.1(12)

Effect: Components are allowed their minimal number of storage units as provided for by their own representation and/or packing.

Floating-point components are aligned on storage-unit boundaries, either 4 bytes or 8 bytes, depending on digits.

Use: Pragma PACK is used to reduce storage size. This can allow records and arrays, in some cases, to be passed by value instead of by reference.

Size reduction usually implies an increased cost of accessing components. The decrease in storage size may be offset by increase in size of accessing code and by slowing of accessing operations.

F.1.9 Pragmas SYSTEM_NAME, STORAGE_UNIT, MEMORY_SIZE, SHARED

These pragmas are not supported and are ignored.

F.1.10 Pragma OPTIMIZE

Pragma OPTIMIZE is ignored; optimization is always enabled.

F.2 Implementation-dependent Attributes

This section describes the name and the type of every implementation-dependent attribute.

There are no implementation defined attributes. These are the values for certain language-defined, implementation-dependent attributes:

```
Type INTEGER.
     INTEGER'SIZE                        = 32 -- bits.
     INTEGER'FIRST                       = -(2**31)
     INTEGER'LAST                        =  (2**31-1)


Type SHORT_FLOAT.
     SHORT_FLOAT'SIZE                    = 32 -- bits.
     SHORT_FLOAT'DIGITS                  = 6
     SHORT_FLOAT'MANTISSA                = 21
     SHORT_FLOAT'EMAX                    = 84
     SHORT_FLOAT'EPSILON                 = 2.0**(-20)
     SHORT_FLOAT'SMALL                   = 2.0**(-85)
     SHORT_FLOAT'LARGE                   = 2.0**84
     SHORT_FLOAT'MACHINE_ROUNDS          = false
     SHORT_FLOAT'MACHINE_RADIX           = 16
     SHORT_FLOAT'MACHINE_MANTISSA        = 6
     SHORT_FLOAT'MACHINE_EMAX            = 63
     SHORT_FLOAT'MACHINE_EMIN            =-64
     SHORT_FLOAT'MACHINE_OVERFLOWS       = false
     SHORT_FLOAT'SAFE_EMAX               = 252
     SHORT_FLOAT'SAFE_SMALL              = 16#0.800000#E-63
     SHORT_FLOAT'SAFE_LARGE              = 16#0.FFFFF8#E63

Type FLOAT.
     FLOAT'SIZE                          = 64 -- bits.
     FLOAT'DIGITS                        = 15
     FLOAT'MANTISSA                      = 51
     FLOAT'EMAX                          = 204
     FLOAT'EPSILON                       = 2.0**(-50)
     FLOAT'SMALL                         = 2.0**(-205)
     FLOAT'LARGE                         = (1.0-2**(-51))*2.0**204
     FLOAT'MACHINE_ROUNDS                = false
     FLOAT'MACHINE_RADIX                 = 16
     FLOAT'MACHINE_MANTISSA              = 14
     FLOAT'MACHINE_EMAX                  = 63
     FLOAT'MACHINE_EMIN                  = -64
     FLOAT'MACHINE_OVERFLOWS             = false
```

```
    FLOAT'SAFE_EMAX                 = 252
    FLOAT'SAFE_SMALL                = 16#0.80000000000000#E-63
    FLOAT'SAFE_LARGE                = 16#0.FFFFFFFFFFFFE0#E63

Type DURATION.
    DURATION'DELTA                  =  2.0**(-14) -- seconds
    DURATION'FIRST                  = - 86,400
    DURATION'LAST                   =  86,400
    DURATION'SMALL                  =  2.0**(-14)

Type PRIORITY.
    PRIORITY'FIRST                  = -127
    PRIORITY'LAST                   =  127
```

F.3 Package SYSTEM

```
package SYSTEM is

   type ADDRESS is private;    -- "=", "/=" defined implicitly;
   type NAME is (UTS, MVS, CMS, Sperry1100, MIL_STD_1750A);

SYSTEM_NAME : constant NAME := UTS ;  -- Target dependent

   STORAGE_UNIT : constant := 8;
MEMORY_SIZE : constant := 2**24;  -- 2**31 for XA mode
          -- In storage units

          -- System-Dependent Named Numbers:

   MIN_INT : constant := INTEGER'POS(INTEGER'FIRST);
   MAX_INT : constant := INTEGER'POS(INTEGER'LAST);
   MAX_DIGITS : constant := 15;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA : constant := 2.0**(-31);
   TICK : constant := 1.0;

          -- Other System-Dependent Declarations

   subtype PRIORITY is INTEGER range -127..127;

-----------------------------------------------------------
-- Implementation-dependent additions to package SYSTEM --
-----------------------------------------------------------

   NULL_ADDRESS : constant ADDRESS;
          -- Same bit pattern as "null" access value
          -- This is the value of 'ADDRESS for named numbers.
          -- The 'ADDRESS of any object which occupies storage
          --  is NOT equal to this value.

   ADDRESS_SIZE : constant := 32;
          -- Number of bits in ADDRESS objects,
          --          == ADDRESS'SIZE, but static.

   ADDRESS_SEGMENT_SIZE : constant := 2**24;
          -- Number of storage units in address segment

   type ADDRESS_OFFSET is new INTEGER;
          -- Used for address arithmetic
```

```
type ADDRESS_SEGMENT is new INTEGER;
        -- Always zero on targets with
        --     unsegmented address space.

subtype NORMALIZED_ADDRESS_OFFSET is
  ADDRESS_OFFSET range 0 .. ADDRESS_SEGMENT_SIZE - 1;
        -- Range of address offsets returned by OFFSET_OF

function "+"(addr : ADDRESS; offset : ADDRESS_OFFSET)
                    return ADDRESS;
function "+"(offset : ADDRESS_OFFSET; addr : ADDRESS)
                    return ADDRESS;
        -- Provide addition between addresses and
        --  offsets.  May cross segment boundaries on targets
        --  where objects may span segments.
        -- On other targets, CONSTRAINT_ERROR will be raised
        --  when OFFSET_OF(addr) + offset not in
        --  NORMALIZED_ADDRESS_OFFSET.

function "-"(left, right : ADDRESS) return ADDRESS_OFFSET;
        -- May exceed SEGMENT_SIZE on targets where objects
        --  may span segments.
        -- On other targets, CONSTRAINT_ERROR
        --  will be raised if
        --            SEGMENT_OF(left) /= SEGMENT_OF(right).

function "-"(addr : ADDRESS; offset : ADDRESS_OFFSET) return
  ADDRESS;
        -- Provide subtraction of addresses and offsets.
        -- May cross segment boundaries on targets where
        --  objects may span segments.
        -- On other targets, CONSTRAINT_ERROR will be raised whe
        --  (OFFSET_OF(addr) - offset)
        --   not in NORMALIZED_ADDRESS_OFFSET.

function OFFSET_OF (addr : ADDRESS)
                    return NORMALIZED_ADDRESS_OFFSET;
        -- Extract offset part of ADDRESS
        -- Always in range 0..seg_size - 1

function SEGMENT_OF (addr : ADDRESS) return ADDRESS_SEGMENT;
        -- Extract segment part of ADDRESS
        -- (zero on targets with unsegmented address space)

function MAKE_ADDRESS (offset : ADDRESS_OFFSET;
                       segment : ADDRESS_SEGMENT := 0)
```

```
                                    return ADDRESS;
            -- Build address given an offset and a segment.
            -- Offset may be > seg_size on targets where objects
            --  may span segments, in which case it is equiv
            --  to "MAKE_ADDRESS(0,segment) + offset".
            -- On other targets, CONSTRAINT_ERROR will be raised
            --  when offset not in NORMALIZED_ADDRESS_OFFSET.

   type Supported_Language_Name is (  -- Target dependent
         -- The following are "foreign" languages:
      ASSEMBLER,
      AIE_ASSEMBLER -- NOT a "foreign" language - uses AIE RTS
   );
         -- Most/least accurate built-in integer and float types

   subtype LONGEST_INTEGER is STANDARD.INTEGER;
   subtype SHORTEST_INTEGER is STANDARD.INTEGER;

   subtype LONGEST_FLOAT is STANDARD.FLOAT;
   subtype SHORTEST_FLOAT is STANDARD.SHORT_FLOAT;

private

   type ADDRESS is access INTEGER;
         -- Note: The designated type here (INTEGER) is
         --       irrelevant.  ADDRESS is made an access type
         --       simply to guarantee it has the same size as
         --       access values, which are single addresses.
         --       Allocators of type ADDRESS are NOT meaningful.

   NULL_ADDRESS : constant ADDRESS := null;

end SYSTEM ;
```

F.4 Representation Clauses

This section describes the list of all restrictions on representation clauses.

"*NOTE: An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware.... If a program contains a representation clause that is not accepted [by the compiler], then the program is illegal.*" (LRM 13.1(10)).

There are no restrictions except as follows:

a. Length clauses are not allowed.

b. Address clauses are not allowed.

c. Record-representation-clause:

   Alignment clauses are not supported and the use of an alignment clause will cause a compile-time error.

   Within a record-representation-clause, the object being represented must be no larger than one 32-bit word.

   The range of bits specified must be in the range of 0..31.

   Record components, including those generated implicitly by the compiler, whose locations are not given by the representation-clause, are layed out by the compiler following all the components whose locations are given by the representation-clause. Such components of the invariant part of the record are allocated to follow the user-specified components of the invariant part, and such components in any given variant part are allocated to follow the user-specified components of that variant part.

F.5 Implementation-dependent Components

This section describes the conventions used for any implementation-generated name denoting implementation-dependent components.

There are no implementation-generated names denoting implementation-dependent (record) components, although there are, indeed, such components. Hence, there is no convention (or possibility) of naming them and, therefore, no way to offer a representation clause for such components.

NOTE: Records containing dynamic-sized components will contain (generally) unnamed offset components which will "point" to the dynamic-sized components stored later in the record. There is no way to specify the representation of such components.

F.6 Address Clauses

This section describes the interpretation of expressions that appear in address clauses, including those for interrupts.

Address clauses are not allowed.

F.7 Unchecked Conversions

This section describes any restrictions on unchecked conversions.

The source and target must both be of a staticly sized type (other than a discriminated record type) and both types must have the same static size.

F.8 Input-Output

This section describes implementation-dependent characteristics of the input-output packages.

(a) Declaration of type Direct_IO.Count? [14.2.5]
0..Integer'last;

(b) Effect of input/output for access types?
Not meaningful if read by different program invocations

(c) Disposition of unclosed IN_FILE files at program termination? [14.1(7)]
. Files are closed.

(d) Disposition of unclosed OUT_FILE files at program termination? [14.1(7)]
Files are closed.

(e) Disposition of unclosed INOUT_FILE files at program termination? [14.1(7)]
Files are closed.

(f) Form of, and restrictions on, file names? [14.1(1)]
UTS filenames

(g) Possible uses of FORM parameter in I/O subprograms? [14.1(1)]
The image of an integer specifying the UTS file protection on CREATE.

(h) Where are I/O exceptions raised beyond what is described in Chapter 14? [14.1(11)]
None raised.

(i) Are alternate specifications (such as abbreviations) allowed for file names? If so, what is the form of these alternatives? [14.2.1(21)]
No.

(j) When is DATA_ERROR *not* raised for sequential or direct input of an inappropriate ELEMENT_TYPE? [14.2.2(4), 14.2.4(4)]
When it can be assigned without CONSTRAINT_ERROR to a variable of ELEMENT_TYPE.

(k) What are the standard input and standard output files? [14.3(5)]
UTS standard input and output

(l) What are the forms of line terminators and page terminators? [14.3(7)]
Line terminator is ASCII.LF (line feed);
page terminator is ASCII.FF (form feed)

(m) Value of Text_IO.Count'last? [14.3(8)]
integer'last

(n) Value of Text_IO.Field'last? [14.3.7(2)]
integer'last

(o) Effect of instantiating ENUMERATION_IO for an integer type? [14.3.9(15)]
The instantiated Put will work properly, but the instantiated Get will raise Data_Error.

(p) Restrictions on types that can be instantiated for input/output?
Neither direct I/O nor sequential I/O can be instantiated for an unconstrained array type or for an unconstrained record type lacking default values for its discriminants.

(q) Specification of package Low_Level_IO? [14.6]
Low_Level_IO is not provided.

F.9 Tasking

This section describes implementation-dependent characteristics of the tasking run-time packages.

Even though a main program completes and terminates (its dependent tasks, if any, having terminated), the elaboration of the program as a whole continues until each task dependent upon a library unit package has either terminated or reached an open terminate alternative. See LRM 9.4(13).

F.10 Other Matters

This section describes other implementation-dependent characteristics of the system.

a. Package Machine_Code
   Will not be provided.

b. Order of compilation of generic bodies and subunits (LRM 10.3:9):
   Body and subunits of generic must be in the same compilation as the specification if instantiations precede them (see AI-00257/02).

F.11 Compiler Limitations

(a) Maximum length of source line?

255 characters.

(b) Maximum number of "use" scopes?

Limit is 50, set arbitrarily by SEMANTICS as maximum number of distinct packages actively "used."

(c) Maximum length of identifier?

255 characters.

(d) Maximum number of nested loops?

24 nested loops.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..254 =>'A', 255 =>'1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..254 =>'A', 255 =>'2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..127 =>'A', 128 =>'3', 129..255 =>'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..127 =>'A', 128 =>'4', 129..255 =>'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..252 => '0', 253..255 =>"298")

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 =>'0', 250..255 =>"69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..100 =>'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..154 =>'A', 155 =>'1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..235 =>' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD-CHARS^#./%!X
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	WILDCARDS*DONT/MATTER
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	90_000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-CHARAC/TER
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	NO/MUCH-TOO-LONG-NAME-FOR-A-FILE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2_147_483_648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2_147_483_647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2_147_483_647 + 1
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-90_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_647 + 1

TEST PARAMETERS

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 =>"2:", 3..252 =>'0', 253..255 =>"11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 =>"16:", 4..251 =>'0', 252..255 =>"F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..254 =>'A', 255 =>'"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2_147_483_648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_OTHER_PREDEF_NUM_TYPE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);". The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.